FLYWEIGHT PATTERN *(TEAM Delulu)*

INTRODUCTION, HISTORY, & OBJECTIVES(TABOADA)

- Similar techniques were used in other systems as early as 1988.
- Coined by *Paul Calder* and *Mark Linton* in 1990 for handling glyph info in document editors.
- Then it was formally described in the "Design Patterns" book by the Gang of Four in 1995.
- It was presented as a way to manage a large number of fine-grained objects efficiently by separating intrinsic and extrinsic state.

UNDERSTANDING FLYWEIGHT PATTERN (PANTOJA)

- A structural design pattern & **Optimization Pattern**
- Reduce & Reuse - based on the idea of *minimizing* memory consumption by *sharing* as much as possible between related objects that are required by the program at the run-time.

KEY COMPONENTS (TABOADA)

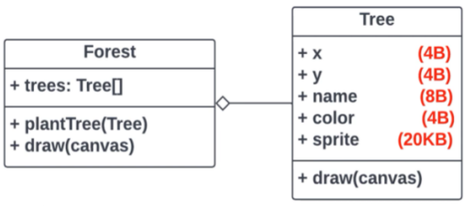
- **INTRINSIC** Attributes/Properties
  - Common among objects
  - Properties that do not change with more instances
  - Can be reused (thus saving more memory)
- **EXTRINSIC** Attributes/Properties
  - Properties that change
- **FACTORY PATTERN**
  - Where we cache/store the data(intrinsic attributes)
  - Objects cached by Factory pattern are called **FLYWEIGHTS**

HOW IS IT BENEFICIAL / USAGE(PANTOJA)

- Ideal for scenarios with a large number of similar objects with attributes that can be reused and is used to optimize RAM usage and enhance performance.
- ***Examples :***
  - Video Games (with a lot of repeating objects - racing games, etc.)
  - Database systems (connection pooling)
  - Web servers (handling client connections)
  - Graphic design software (shared graphical elements)
  - E-commerce Apps (listing and selling many products/items)

SITUATIONAL SAMPLE PROBLEM (DATAN)

- Scenario:
  - We want to create a program that renders different types of trees onto a canvas, effectively representing a forest
- Problem:
  - RAM per Tree(object) is ~21KB
  - What if we were to render 1 Million Trees?
    THIS WILL CAUSE MEMORY PROBLEM
- Solution:
  - Implement **FLYWEIGHT PATTERN:**
    1. Identify Extrinsic and Intrinsic states
    2. Pull out intrinsic state of object and store into new object class
    3. In Forest(main) class, we have to add an array of TreeType(flyweight) references whose index stays in sync with our array of Trees(object)



RAM Calculation:

$$4B + 4B + 8B + 4B + 20KB = \sim 21KB$$

$$1,000,000\ Trees * 21KB = 21GB$$

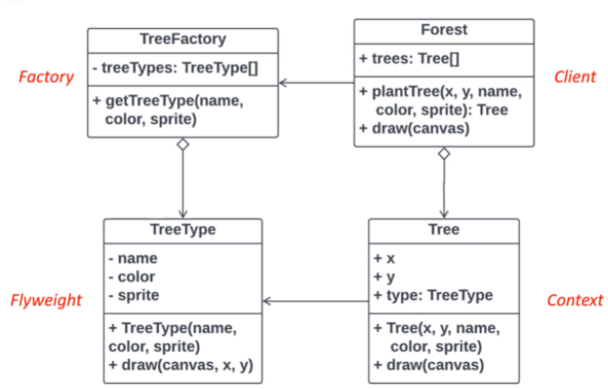
THIS APPROACH CAN DRASTICALLY REDUCE OUR MEMORY FOOTPRINT

RAM Calculation:

Extrinsic
$4B + 4B = 8B$

Intrinsic
$8B + 4B + 20KB = \sim 21KB$

1,000,000 Trees * 8B +
3 TreeTypes * 21KB = 8.1MB



FLYWEIGHT PATTERN *(TEAM Delulu)*

FLYWEIGHT PATTERN CODE SNIPPET

A. Ball.java (PANTOJA)

```java
package flyweightexample;

public class Ball {

    private String color;
    private String imageURL;
    private int coordX;
    private int coordY;
    private int radius;

    public Ball(String color, String imageURL){
        this.color = color;
        this.imageURL = imageURL;
    }

    public void setX(int x) {
        this.coordX = x;
    }

    public void setY(int y) {
        this.coordY = y;
    }

    public void setRadius(int radius) {
        this.radius = radius;
    }

    public void draw() {
        //logic to  draw on canvas
    }
}
```

B. Main.java(TABOADA)

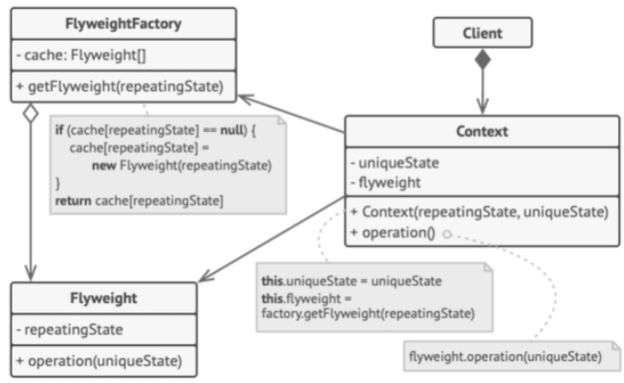
```java
public class FlyweightDemo {
    private static final String colors[] = { "Green", "Yellow", "Pink"};
    private static final Map<String, String> urlMap = Map.ofEntries(
            new AbstractMap.SimpleEntry<String, String>("Green", "url1"),
            new AbstractMap.SimpleEntry<String, String>("Yellow", "url2"),
            new AbstractMap.SimpleEntry<String, String>("Pink", "url3")
    );

    public static void main(String[] args) {
        for(int i=0; i < 10; ++i) {
            String color = getColor();
            String url = urlMap.get(color);
            Ball ball = (Ball)BallFactory.getBall(color, url);
            ball.setX(getX());
            ball.setY(getY());
            ball.setRadius(75);
            ball.draw();
            System.out.println(ball.hashCode());
        }
    }
    private static String getColor() {
        return colors[(int)(Math.random()*colors.length)];
    }

    private static int getX() {
        return (int)(Math.random()*50 );
    }

    private static int getY() {
        return (int)(Math.random()*50);
    }
}
```

UML/CLASS DIAGRAM OF FLYWEIGHT PATTERN (DATAN)



C. BallFactory.java(DATAN)

```java
package flyweightexample;

import java.util.HashMap;

public class BallFactory {
    private static final HashMap ballMap = new HashMap();

    public static Ball getBall(String color, String url) {
        StringBuilder sb = new StringBuilder();
        sb.append(color);
        sb.append(url);
        String ballCacheKey = sb.toString();
        Ball ball = (Ball)ballMap.get(ballCacheKey);

        if(ball == null) {
            ball = new Ball(color, url);
            ballMap.put(ballCacheKey, ball);
            System.out.println("Creating circle of color : " + color);
        }
        return ball;
    }
}
```

PROS(PANTOJA)

- Memory Efficiency
- Performance Improvement
- Simplified Code
- Versatile Application

CONS(PANTOJA)

- Complexity
- Reduced Object Independence
- Increased Initialization Overhead
- Limited Applicability

SUMMARY(ALL)

In conclusion, the Flyweight pattern stands as a powerful design solution for optimizing memory usage and enhancing performance in scenarios where numerous fine-grained objects are at play. By efficiently managing intrinsic and extrinsic states while promoting data sharing, this pattern provides a practical approach to tackle resource-intensive tasks in software development. Its ability to reduce redundancy and minimize memory overhead makes it an invaluable tool for creating more efficient and responsive applications across various domains, from graphics rendering to document editing, ultimately contributing to improved user experiences and resource utilization.